Published in the IEEE Proceedings of the *International Conference on Parallel Computing in Electrical Engineering (PARELEC 2000),* 27-30 August 2000, Trois-Rivieres, Quebec CA, pp. 13-17.

C++2MPI: A Software Tool for Automatically Generating MPI Datatypes from C++ Classes.

Dr. Roger Hillson
Naval Research Laboratory
email: hillson@ait.nrl.navy.mil

Dr. Michal Iglewski
University of Quebec at Hull
email: iglewski@uqah.uquebec.ca

ABSTRACT

The Message Passing Interface 1.1 (MPI 1.1) standard defines a library of message-passing functions for parallel and distributed computing. We have developed a new software tool called C++2MPI which can automatically generate MPI derived datatypes for a specified C++ class. C++2MPI can generate data types for derived classes, for partially and fully-specialized templated classes, and for classes with private data members. Given one or more user-provided classes as input, C++2MPI generates, compiles and archives a function for creating the MPI derived datatype. When the generated function is executed, it builds the derived MPI datatype if the datatype does not already exist, and returns the value of an MPI handle for referencing the datatype. PGMT (Processing Graph Method Tool) is a set of application program interfaces for porting the Processing Graph Method (PGM), a parallel programming method, to diverse networks of processors. C++2MPI was developed as a component of PGMT, but can be used as a stand-alone tool.

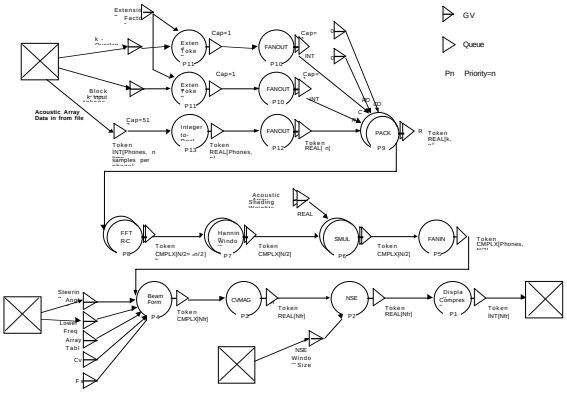


Figure 1 - PGM Dataflow Graph [1]

1: Introduction to PGM and PGMT

The *Processing Graph Method* (PGM) is a language-independent data-flow method developed at the Naval Research Laboratory (NRL) [1, 2]. The intent of the PGM approach is to reduce development time for parallel and distributed programs and to increase application portability across diverse platforms. PGM enables the application-developer to specify his application as a data-flow graph, an approach which exposes the parallelism inherent in the program [Figure 1]. A PGM graph consists of *nodes* connected by directed *arcs* which specify the path and direction of data flow within an application. A PGM node is either a *place* or a *transition*. *Transitions* are the computational elements of the graph, while the *places* store data. The data are manipulated in the form of *tokens*, which are packets of strongly-typed data. When a transition executes, it will typically read and consume a token from one or more upstream places, and produce one or more tokens to each of its output places.

1.1: PGM Families and Base Types.

The fundamental PGM data structure conveyed by tokens is called a *family* [2]. A family is a hierarchical data structure of elements of some specified *base-type*. A base type is also called *a leaf-node* type. The base type cannot be a pointer, but will typically be a predefined datatype in the high-order language (HOL) chosen for a specific implementation of PGM. In C or C++, for example, the base type of a family could be *int*, *float*, or *long*. A user may also define his own homogeneous or heterogeneous PGM base type as an aggregate data structure supported by the HOL. The data structure could be a C structure, a C++ class, or a Pascal record.

1.2: PGMT

PGM has been implemented on both serial and parallel computer architectures, but the cost of porting PGM to a new computer architecture is high. The *Processing Graph Method Tool* (PGMT*)* is a set of software application program interfaces (APIs) designed to greatly reduce the cost of porting PGM to a new computer system [1]. PGMT is written in C++, and uses many of the functions of the C++ standard template library (STL). PGMT includes a PGM *Graphic User Interface* (GUI); a *Graph State File* (GSF) for saving both the graph layout and the PGM graph parameters; a GSF-to-C++ translator; an MPI-based communications library (referred to as middleware); decomposition functions for partitioning the graph into concurrent processes; and runtime functions for scheduling transition executions, and for reassigning transitions dynamically to different processors.

1.3 PGMT and MPI 1.1

The PGMT communication library is based on the *Message Passing Interface 1.1* (MPI 1.1) standard [3]. MPI 1.1 has been widely implemented on diverse parallel processors, thus facilitating PGMT portability across platforms. Although PGMT testing has been conducted primarily on a distributed network of Sun and SGI workstations, we have also tested components of the PGMT communications library on an SGI Origin 2000, and a Hewlett-Packard Convex Exemplar. PGMT is implemented in C++, although PGM proper is language independent.

The MPI communication functions (e.g. *send*, *receive*, *broadcast*, *scatter*, *gather*, *global maximum*, *global minimum*) are strongly typed: each function call requires an explicit argument specifying the MPI datatype being sent or received. MPI provides built-in datatypes for the common scalar arithmetic types such as *int* [i.e. MPI_INT] and *float* [i.e MPI_FLOAT]. MPI also enables a user to define his own MPI datatype as a composite of one or more rudimentary MPI datatypes. A derived MPI datatype is an opaque object comprised of a sequence of ordered pairs of (*i*) either basic *or* derived datatypes and (*ii*) integral byte displacements from the beginning of a hypothetical send buffer [see 3, sect. 3.12]. MPI provides functions for building derived MPI datatypes, and for assigning values to the MPI *handles* which reference them. The integer displacements are specified as offsets into an object instantiated from a user-defined C structure.

As discussed in Section 1.1, PGM permits a user to define his own PGM family base types. PGMT is written in C++. In the PGMT environment, users can define their own PGM base types by writing C++ classes. An MPI datatype must be created for each user-defined C++ class. The tool C++2MPI automates the process of creating an MPI datatype for a given C++ class.

2: Automatic MPI Datatype Generation

2.1: Previous work.

The team of J. E. Devaney at National Institute of Standards and Technology developed two tools [4, 5] that facilitate the use of data structures in MPI. AutoMap is a utility that creates MPI types from C data structures. AutoLink is an MPI library that automates two functions: the generation of MPI data types, and the sending and receiving of dynamic data structures. The MPI data types are generated by running AutoMap on the program code.

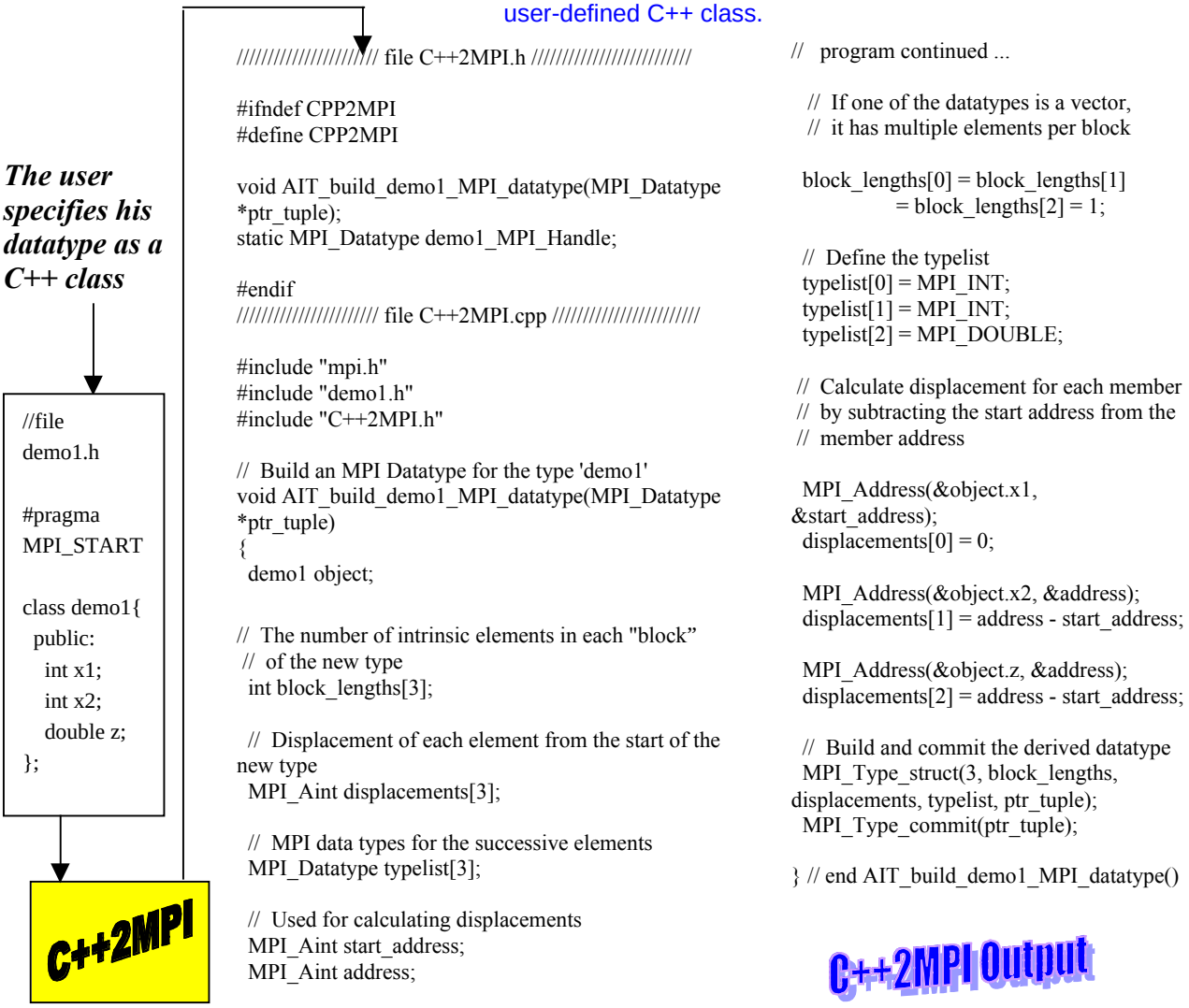
2.2: C++2MPI

C++2MPI is a tool that creates MPI data types from C++ data structures. In C++, a *pragma* is an implementation-dependent preprocessing directive. A user requests an MPI datatype for a user-defined class by inserting the pragma MPI_START in front of the class definition, and the pragma MPI_END at the end. MPI_END can be omitted if the end of the class definition coincides with the end of the file.

The tool creates two files, C++2MPI.h and C++2MPI.a. If an MPI datatype is requested for a user-defined datatype, then the file C++2MPI.h will contain a prototype (i.e. declaration) for the generated function which will build the derived datatype. C++2MPI.h will also include a declaration for a variable of type *MPI_Datatype*. The address of this variable will be passed as an argument to the function, and initialized as a handle to the derived datatype when the function executes. The output file C++2MPI.a is a library containing the compiled functions for the prototypes defined in C++2MPI.h.

The data members of a user-defined type for which an MPI datatype is requested may be (*i*) any predefined C datatype which has a corresponding MPI predefined data type, (*ii*) a different user-defined data type, or (*iii*) an array of the datatypes mentioned in (*i*) or (*ii*). Pointer types and static data members are not allowed. Figure 2 illustrates sample input and output files for C++2MPI. C++2MPI.cpp contains the function for building the derived datatype; this function is compiled and archived within C++2MPI.a.

Figure 2: Automatic generation of MPI Datatypes for a user-defined C++ class.



2.3 Implementation of C++2MPI.

C++ has an ambiguous grammar in the sense that LR(k) analysis for any fixed value of k is not sufficient to remove the ambiguity. One example is the "typedef vs. identifier" conflict. A standard solution to this problem is to give the lexical phase access to the C++ symbol table. Another example is the "declaration vs. expression" ambiguity which requires an arbitrary amount of look-ahead. These problems and the template construct make creating a C++ parser using the traditional YACC/LEX approach a very difficult task [6]. C++2MPI is a modified version of John Lilley's C++ parser. Lilley's parser uses the Purdue Compiler Construction Tool Set (PCCTS). Both Lilley's parser, PCCTS and the C++ grammar conceived in conjunction with NeXT Computer Inc. are in the public-domain.

PCCTS [7] is a set of software tools which facilitates the implementation of compilers and other translation systems. These tools currently include ANTLR (Another Tool for Language Recognition) and DLG (DFA-based Lexical analyzer Generator). In many ways, PCCTS is similar to a highly integrated version of YACC [8] and LEX [9], where ANTLR (ANother Tool for Language Recognition) corresponds to YACC and DLG (DFA-based Lexical analyzer Generator) functions like LEX. Both tools generate parsers from a BNF-like grammar description. However, PCCTS has many additional features which make it easier to use for a wider range of translation problems. PCCTS grammars contain specifications for lexical and syntactic analysis, intermediate-form construction, and error reporting. Rules may employ *Extended Backus Naur Form* (EBNF) grammar constructs and may also define parameters, return values and local variables. Languages described in PCCTS are recognized via

Strong LL(k) parsers constructed in pure, human-readable, C code. As a result, PCCTS compilers can be traced and debugged with standard C tools.

The parser developed by John Lilley is based on a C++ grammar written by S. Srinivasan, T. Parr, and R. Quong. J. Lilley added some extra features such as a full ANSI preprocessor, complete type and declaration information, nested scopes, and STL-based symbol tables. The basic difference between the C++2MPI tool and Lilley's parser is the creation of the MPI class as a super-class of the parser, scanner and token classes in Lilley's parser.

The core of the C++2MPI tool is a function which builds the MPI datatypes denoted by user-defined C++ classes. This function should be invoked for any class or instantiation of a templated class for which a corresponding MPI data type is needed. Since Lilley's parser reparses the class template for any instantiation of the templated class, it is sufficient to invoke this function for any class-specifier in the C++ source code. The function has two arguments. The first is a reference to the type describing a requested class and the second is a reference to the scope in which the class is declared. Generating the body of this function requires access to the list of its base classes, the list of its members and their types. All this information can be found by using different operators from the symbol table built by the parser.

2.3.1: Special cases

- access to private data members In order to create an MPI data type, the displacement of each data member in the user-defined type must be calculated. This is done by using the MPI_address function. To get access to private and protected members, the tool makes all memory specifiers public in a copy of the code which is used to create an archive. This has no impact on the user's source code since any access violation will be detected by the compiler.

- templates For *each* instantiation of a templated class, C++2MPI generates a function to build the corresponding MPI datatype. For example:

```
// User-defined templated class
#pragma MPI_START template <class T> class Nemo
{public: T x; T y;};

// User-defined instantiations of templated classes
typedef Nemo<int> NI;
typedef Nemo<float> NF;

// Prototypes for functions generated by C++2MPI
void build_Nemo_int_MPI_datatype(MPI_Datatype
*ptr_tuple) { /* ... */ };
void build_Nemo_float_MPI_datatype(MPI_Datatype
*ptr_tuple){ /* ... */ };
```

- derived classes In order to build an MPI datatype for a derived class, it is necessary to build an MPI datatype for the base class itself. One member of the derived class will represent the base class members. The displacement of this member is computed with a dynamic cast. The following partial example demonstrates this approach:

USER-DEFINED DERIVED C++ CLASS:

```
#pragma MPI_START

class base_v {
public:
 int x1;
 base_v(int in1 = 0) { x1 = in1; }
};

class derived_v : public base_v {
public:
  int y1;
  double z1;
derived_v(int in1 = 0, int in2 = 0, double in3 = 0.0) :
base_v(in1) {
 y1 = in2;
 z1 = in3;  } };
```

C++2MPI-GENERATED FUNCTIONS FOR CREATING THE MPI DATATYPES :

```
  // I. Function to build an MPI Datatype 'base_v'
void build_base_v_MPI_datatype(MPI_Datatype
*ptr_tuple)   { // ... }

 // II. Function to build an MPI Datatype 'derived_v'
void AIT_build_derived_v_MPI_datatype(MPI_Datatype
*ptr_tuple) {

derived_v object;

 // ...
 // IIa. Define the typelist
 // ...
build_base_v_MPI_datatype(&base_v_MPI_Type);
typelist[2] = base_v_MPI_Type;

// IIb. Calculate displacement for each member by
// subtracting the start address from the member address
```

```
MPI_Address(&object.y1, &start_address);
displacements[0] = 0;
// ...
MPI_Address(dynamic_cast(&object), &address);
displacements[2] = address - start_address;
// ... }
// End derived class example
```

3: PGMT Integration

C++2MPI also generates an auxiliary file which defines an STL vector of structures. There is one structure defined for each user-defined type (i.e. class) provided by the user. Each structure has two components: the *name* of the user-defined type (i.e. class name) and the *address* of the function used to build the corresponding MPI datatype. For example, if a user defines a class named *complex*, the two components of the corresponding structure will be:

```
(1)    string( "complex" );
(2)    &build_complex_MPI_datatype;
```

The first time the function build_complex_MPI_datatype() is called it builds the MPI datatype for the user-defined class complex, and returns an MPI handle of type *MPI_Datatype* [3]. On subsequent calls to build_complex_MPI_datatype, the function returns the correct value of the MPI handle without rebuilding the MPI datatype.

PGMT uses the vector of structures to build an STL associative map. The map keys are the class names, and the values returned are the function addresses. The map also includes functions which return the *predefined* MPI datatypes (handles) for the basic C++ arithmetic types. For example, *int* is the key to a function which returns the value of MPI_INT. The associative map is accessed via a function call generated by the PGMT GSF-to-C++ translator.

4: Summary and Conclusions

The *Message Passing Interface 1.1* (MPI 1.1) standard defines a standard library of message-passing functions for parallel and distributed computing. MPI includes functions for building the user-defined MPI datatypes required for sending and receiving instances of user-defined C structures.

In this paper, we discuss a new software tool called C++2MPI. C++2MPI automatically generates MPI derived datatypes corresponding to C++ *classes*, as well as generating derived datatypes for standard C structures. C++2MPI accepts one or more user-defined C++ classes as inputs, and generates and compiles a C program which will create the corresponding user-defined MPI datatype. An MPI handle to the datatype is returned to the calling program. C++2MPI can generate data types for derived classes, for partially and fully-specialized templated classes, and for classes with private data members.

The Processing Graph Method (PGM) is a data-flow method for programming parallel processors. The integration of C++2MPI with the Processing Graph Method Tool (PGMT) is discussed. PGM and PGMT were developed at the Naval Research Laboratory.

Acknowledgements: We wish to acknowledge the contributions and suggestions of David J. Kaplan, Richard S. Stevens, and David M. Armoza to the work reported in this paper.

References

[1.] Kaplan, David J, "An Introduction to the Processing Graph Method," Presented Monday, March 24, 1997 at the International Conference and Workshop on Engineering of Computer Based Systems Hosted by IEEE Computer Society Technical Committee on Engineering of Computer Based Systems and The University of Arizona Electrical and Computer Engineering Department.
[2.] Kaplan, D., and R. Stevens, "Processing Graph Method 2.0 Semantics," Naval Research Laboratory, 15 Sept. 1995.
See http://www.ait.nrl.navy.mil/pgmt/ for documentation about PGM and the Processing Graph Method Tool.
[3.] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, 12 June 1995.
[4.] J.E. Devaney, M. Michel, J. Peeters, E. Baland, "AutoMap: A Data Structures Compiler for the Automatic Generation of MPI Data Structures Directly From C Code", NIST, 1997.
[5.] M. Michel, D.S. Goujon, J. Peeters, J.E. Devaney, "AutoMap & AutoLink: Tools for communicating complex data-structures using MPI", Proc. CANPC'98.
[6.] J. Lilley, "John Lilley's PCCTS-Based LL(1) C++ Parser", http://www.empathy.com/pccts/cppgrammar_unix.tar.gz
[7.] Terence J. Parr, Russell W. Quong, "ANTLR: A Predicated-LL(k) Parser Generator". Software Practice & Experience 25(7): 789-810 (1995).
[8.] Stephen C. Johnson, "YACC: Yet Another Compiler-Compiler". Bell Laboratories, Murray Hill, NJ, 1978.
[9.] M. E. Lesk, "LEX -- a Lexical Analyzer Generator". CSTR 39, Bell Laboratories, Murray Hill, NJ, 1975.